Available online at www.sciencedirect.com

SCIENCE DIRECT®

Electronic Notes in Theoretical Computer Science

Electronic Notes in Theoretical Computer Science 151 (2006) 77–95

www.elsevier.com/locate/entcs

Applying a Stochastic Model to a Dynamic, QoS Enabled Web Services Hosting Environment

Charles Kubicek[1] ,[2]

North East Regional e-Science Centre
University of Newcastle
UK

Abstract

Data centres which host Web services for other organisations and users in a Grid environment must provide for Quality of Service (QoS) requirements to be specified to ensure deployed services perform as desired. As services hosted by a data centre receive unpredictable rates of demand, servers must be allocated dynamically to service pools that are over utilised to avoid breaking QoS requirements. This work describes how a cost based stochastic model for resource allocation is used in data centre middleware to balance server utilisation, and how the model was used to enable a data centre to meet QoS requirements. The stochastic QoS model is compared to two other QoS models and is shown to be the most effective in a number of experiments.

Keywords: Quality of Service, Dynamic Resource Allocation, Stochastic Modelling.

1 Introduction

As Grid based technologies are becoming more commonplace, data centres which provide hosting services are being used by organisations to run computationally intensive applications and host services. A data centre may negotiate agreements with its customers to provide certain Quality of Service (QoS) levels for deployed services, which are formally specified in a Service Level Agreement (SLA). A SLA provides a guarantee to a data centre customer ensuring that requests from their own customers will receive a certain level of service.

The data centre must cope with an unpredictable demand for each service hosted while ensuring any QoS requirements for deployed services are met. To avoid temporary over-loading of some services and under-loading others, a data centre may

[1] This work was carried out as part of the collaborative project GridSHED (Grid Scheduling and Hosting Environment Development), funded by British Telecom and the North-East Regional e-Science Centre.

[2] Email: charles.kubicek@ncl.ac.uk

doi:10.1016/j.entcs.2006.03.013

dynamically allocate computational resources to services which are experiencing a high demand, and away from those services which have no or little demand.

Our aim is to develop policies for performing such dynamic reallocation efficiently. These policies are based on a stochastic optimisation model studied in [9,6]. That work showed that a certain heuristic switching rule is close to optimal. In the present study we have implemented the heuristic in a real environment. Its performance was measured and compared to too simpler resource allocation policies.

A recently suggested approach to job submission in Grid systems involves deploying applications on a service provider that are then made available as Web services [2,12] which may be invoked by authorised parties using SOAP [11]. We employ this approach of job submission in our system so that incoming jobs are SOAP messages directed to Web services hosted on servers, and each message is seen as one job by the system. Also described is a method of uploading and dynamically deploying Web services into the system based on user demand.

We describe three approaches of extending the system to meet QoS requirements. Firstly we describe how holding cost values used by the heuristic to make reallocation decisions may be manipulated to favour job types with a better QoS. Then we analyse the response times of jobs using a QoS ratio calculation and a QoS percentile calculation to determine if QoS requirements are being met and reallocate servers if they are not. The approaches are tested using an implementation of the system with job submission patters similar to those observed in a real job submission system.

Both the problem of under-utilisation and QoS in data centres has been studied and a number of solutions proposed. A model using load prediction shows how QoS requirements may be met even at peak arrival rates by reconfiguring different parameters [8]. Another model [10] showed that by switching servers between applications in a data centre using arrival rates, service times and CPU utilization measurements, large resource savings may be made. Methods of prediction have also been used to dynamically adjust the resource share of applications [3]. Other work has shown servers may have a new operating system loaded when joining a new server pool [4]. The other reported work differs from the work in this paper as we assume grid-style jobs which are long running and computationally intensive.

2 System architecture

2.1 Resource provisioning model

The proposed resource management system provides dynamic provisioning capabilities for a system model containing a number of servers which together process requests for a number of hosted services, the number of which may change over time as services are deployed and removed. Each SOAP message sent to a service is one job. Jobs arrive and are assigned a job type i depending on the service being requested, where $i = 1, 2, \ldots, m$. Jobs are then routed to a queue associated with the service where the queue size for jobs of type i at a given time is given as j_i. All queues are assumed to be unbounded and jobs currently being serviced are still seen

in a queue. A number of servers k_i are present in a pool dedicated to possessing jobs of type i, where the total number of servers in the system is given as:

$$k_1 + k_2 + \ldots + k_m = N$$

Servers may differ in hardware and software and may have different capacities, but servers of type i must be configured correctly to service jobs of type i. Servers in a pool may be geographically disparate and reside at different sites providing an appropriate network and security infrastructure exists. The intervals between arrivals of job type i are measured and the average arrival rate is given as λ jobs per second. The service time for each job is also measured and the average service rate of job type i is given as μ jobs per second. The average values are evaluated using the measurements of a window of the last x jobs, where x may be defined by a system administrator. A larger value of x will therefore yield a more accurate result at the expense of a longer processing time. We also take into account the cost of holding a job in a queue, and denote by c_i the cost for holding a job of type i for one time unit. This cost may reflect the importance of one job type over another.

While the allocation of servers to pools may be static after a service deployment we are interested in the case where servers may continuously switch between pools. All jobs currently being serviced by a server that is to switch pool must be halted, removed from the server and placed at the front of their queue in a waiting state. A job may be check-pointed if the environment allows and the job state saved, or any previous processing of the job is cancelled. The processing time of a cancelled job is not measured by the system. The time it takes to switch a server from pool a to pool b and is on average equal to $1/\zeta_{ab}$. During that interval the server is unavailable to process jobs of any type. As different jobs may have different hardware and software dependencies a limitation on which servers can belong to some pools may exist. While dynamically modifying hardware to match a destination pool may be very difficult, the dynamic reconfiguration of software during a switch is possible.

2.2 Pooling system

A cluster of servers is partitioned into conceptual pools associated with a service that executes jobs of type i, and a central management component collects real time data from each pool to make reallocation decisions. Each conceptual pool is managed by a Pool Manager which communicates with each server, and a Cluster Manager communicates with each pool. This architecture is shown is figure 1.

`The Node Manager` runs on each server capable of processing jobs and makes adjustments to its host allowing it to switch pools.

`The Pool Manager` and schedules incoming jobs, keeps track of the servers in its pool, and participates in coordinating switches of servers to and from other pools. Queuing and scheduling may be done by using an existing Resource Management System as discussed in section 6.

`The Cluster Manager` dispatches submitted jobs to the appropriate pool based on job type, and stores information about arrival rates to be used in the switching policy. The Cluster Manager receives information from each Pool Manager which

is combined with stored data to make reallocation decisions. The Cluster Manager acts as a coordinator between two pools during a server switch.

Dispatching a job to a pool is a trivial task and requires no scheduling as jobs are queued and scheduled at each Pool Manager, so queues are not likely to build up at the Cluster Manager. The distributed queuing and scheduling amongst Pool Managers helps disperse load over the system. Together the Cluster Manager and Pool Managers constantly monitor events in the system including job arrival times and queuing and processing times of each job type which allows the Cluster Manager to make informed decisions on server switching between pools. The pooling system frees the Cluster Manager of needing to keep updated information about every server in the system to use in reconfiguration decisions, which greatly increases the scalability of the system which is vital in a data centre.

A policy manager component in the Cluster Manager contains policies which make server switching decisions. The policy manager has access to all the real-time and recent data collected by the system and may be invoked at any time to produce a server switching decision, which could of course be no switch. The currently loaded policy may be changed during run-time, and policy related parameters may be altered such as the window size of system measurements used to make server switching decisions.

3 Resource allocation heuristic

To determine when a server should be switched between conceptual pools a resource allocation heuristic policy derived from stochastic mathematical modelling is used [9]. The heuristic policy calculates a close to optimal allocation of N servers to m pools while taking into account switching costs. An optimal switching policy would analyze the current sate of the system in terms of queues, arrival rates and response

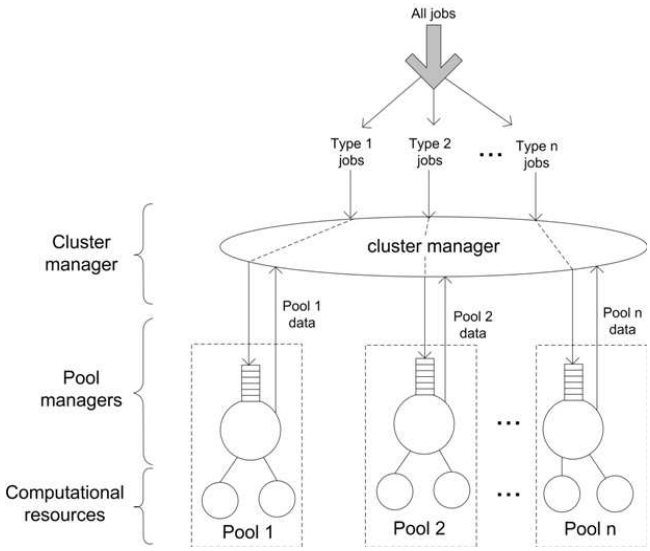


Fig. 1. Pooling architecture

times, and return the optimum server switch decision that could be made at that moment. The optimal policy has been shown to be complex to evaluate, although the heuristic policy has been proved to perform well compared to the optimal policy in simulations. The heuristic policy was developed in conjunction with the middleware and shares some of the same assumptions, particularly assuming long running jobs that will result in queues forming. The heuristic also assumes that one server processes one job at a time which may not be true for servers with more than one processor, and assumes that each server has the same processing capability which is unlikely to be true in a data centre.

The idea behind the heuristic is to calculate a suitably defined cost imbalance V_{ab} between any two pools a and b. A server is switched form pool a to pool b when the corresponding V_{ab} has the largest positive value for any pair of pools. The details of the calculation are shown in figure 2. The reasons for choosing this particular form of V_{ab} are explained in [9,6].

The heuristic has been used in another related system [6] which showed that the heuristic can reallocate resources in a way that reduces response time in a job processing system. In an experiment, 1000 jobs of 3 different job types were submitted to a cluster of 9 servers where 3 server pools processed the jobs. The processing time of the job types was 20, 30 and 40 seconds, and the system received an total offered load of 70%. The load was made up of requests for the three job types with one type constituting 80% of the total load and the other types constituting 10% each. The experiment was performed twice, once with server switching enabled and one with switching disabled, where each job type had a static allocation of 3 servers. The arrival rates of the job types changed after every 100th job submission so each job type constituted 80% of the load a for approximately one third of the experiment. This ensured servers would be switched between pools. Figure 3 shows the results of the experiment in which the response times can be seen to be lower with switching than without switching.

3.1 Heuristic assumptions

To effectively apply the heuristic to data centre middleware certain aspects of the heuristic must be changed, but as the heuristic has proven to make switching decisions very close to the optimal solution it would be in our interests to make as few changes as possible.

The assumption of one server processing one job at a time does not hold in an environment of multi-processor servers. Should the policy assume each server processes one job at a time when in fact some servers processed multiple jobs simultaneously, jobs would appear to be executing faster per server than they actually were. This would then lead to switching decisions based of incorrect knowledge, degrading the performance of the system. One solution is to make sure each server only runs one job at a time which is a waste of resources, so the way the heuristic is applied to the obtained system data is altered.

To apply the heuristic to servers with more than one processor, we supply the heuristic with the number of processors capable of processing jobs in a pool instead

```
1    P_from ← null
2    P_to ← null
3    V ← 0
4    for all i in pairs of job types a and b do
5        V_ab = c_b ( j_b + 1/ζ_{a,b} [λ_b − μ_b min(k_b, j_b)] ) -
           K c_a ( j_a + 1/ζ_{a,b} [λ_a − μ_a min(k_a − 1, j_b)] )
6        if V_ab > V
7            V ← V_ab
8            P_from ← a
9            P_to ← b
10       end if
11   end for
12   if V > 0
13       switch from P_from to pool P_to
14   end if
```

The parameters in the above expression are:

c_i	the holding cost at pool i.
k_i	the number of servers present in pool i.
j_i	the queue size at pool i.
λ_i	the arrival rate at pool i.
1 / μ_i	the average service time for jobs at pool i.
1 / ζ_{ab}	the average switching time of a server from pool a to pool b.
K	a suitable chosen integer to discourage too much switching (e.g. K=5)

Fig. 2. Pseudo code for the resource allocation policy

of the number of servers. When the heuristic makes a switch decision the Pool Manager which is to lose a server is informed of the switch and responds with the number of processors on the server it has chosen to lose. The policy is then recalculated, replacing $k_a - 1$ on line 5 of figure 2 with: $k_a - P$where P is the number of processors on the server which is to be switched. With the new value in the heuristic, the total holding cost at the source pool without the given numbers of processors can be evaluated. An alternative approach to providing the policy with processor data is for the Cluster Manager to keep track of processors within servers and use the policy to workout the best switch without having to communicate with any Pool Managers, although this method is not scalable in a system with many

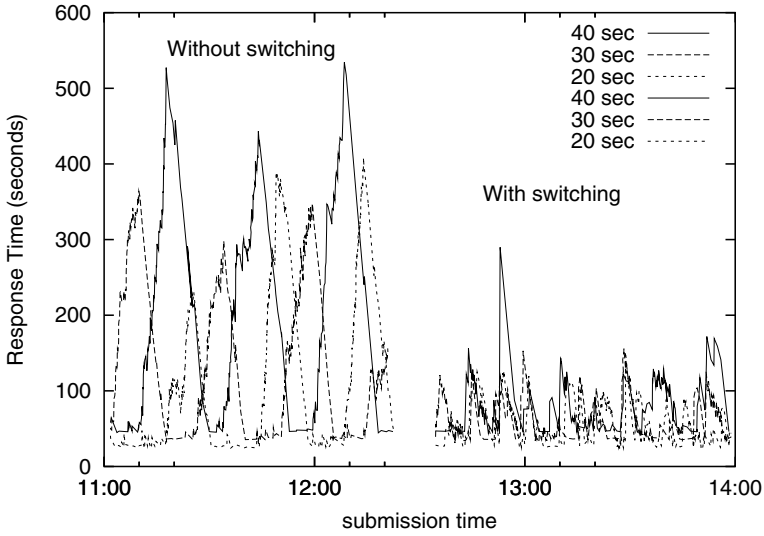


Fig. 3. Response times for three different job types in a system with 9 servers, with and without server reallocation.

pools and servers with multiple processors and is not in accordance with the original aim of keeping the Cluster Manager from having to store and process individual server data.

Another assumption that does not hold is that all servers have the same processing capabilities. While this is not correct, the model measures the processing time of each job and calculates an average in a given window to determine a value for use in the heuristic. The calculating of this average could even out any relative difference in processor speed.

4 QoS metrics and enforcement

A data centre that provides QoS must specify which QoS metrics it enforces, and how those metrics are specified and measured. The QoS delivered by a data centre is perceived by both the customers who have deployed their applications as services, and by users invoking those services. We will focus on providing QoS options for the customers deploying their applications by developing resource allocation policies, and we do not go into the details of SLAs and formal QoS specification. Two QoS metrics likely to be of interest are listed below.

`Average response time;` the measured time between a job arriving and departing the system, essentially the sum of queuing time and processing time in seconds.

`Response time percentile bounds;` specifies that a given percentage of jobs must complete within a given time. For example, the requirement may be that 95% of response times are less than 5 seconds.

An average of the response time is used because it is impossible for a customer or data centre to guarantee every job will always meet a given time requirement, but

it is more reasonable to specify an average over a period of time. Using a percentile is a more strict measurement as a high number of jobs must complete within a time requirement, where as with the average response time measurement a larger set of jobs which have just failed to meet the target can be compensated by a smaller set of jobs which have met the target well inside the target time.

A moving window may be used to monitor the last x jobs to complete execution. An average may then be taken from the window, as may the percentage of jobs which violated the percentile bounds. If when calculating the average response time the value of x is relatively small (e.g. 5 jobs) then the system may react quickly to a change in the response times, however if the value of x is too small the average may not be accurate enough for the system to make a correct decision. When calculating the response time percentile bounds a larger window may be necessary (e.g. 20 jobs) to ensure the system does not react in an extreme fashion due a high percentage of jobs failing the requirement, when in fact only one job took too long to complete.

Our aim is to provide for response-time QoS for Web services by allocating servers based on QoS targets, as well as other system information if necessary. Three QoS models are proposed, the first two measure and react to observed QoS performance and the last uses the resource allocation heuristic. We assume that a QoS policy applies to all pools (sets of pools cannot have different policies), and for each pool an appropriate QoS value has been set, even if it is a default value.

4.1 QoS Policy 1: Average Response Time policy

The first policy switches servers according to the QoS performance measured by the relative difference between the average measured service time and target response time, and switches a server from the pool with the highest ratio to the pool with the lowest. The difference between the measured and target response time is expressed as a deviation [8] which may be positive or negative, the response time is monitored using a window of the last x jobs to complete, and the size of the job window may vary as described earlier. Deviations may be measured and acted upon anytime and if a negative deviation exists, a server is switched from a pool with a high positive deviation (only if one exists) to pool with a negative deviation. As in the original resource allocation model one machine may be switched at a time, and if the deviation is calculated often multiple machines will arrive in a pool within a short amount of time to deal with negative QoS deviations. The monitoring only approach does not try to analyse queue lengths so a rise in service demand will only be observed after response times have increased. The pseudo code for the calculation is shown in figure 4.

4.2 QoS Policy 2: Percentile Policy

The Percentile Policy measures the percentage of jobs for each pool that have met their response time target, and makes a server switch from a pool with the highest percentage of to a pool with the lowest percentage. The pseudo code used to determine a server switch via the Percentile Policy is shown in figure 5. We assume

```
1   R_max ← −∞
2   R_min ← ∞
3   P_max ← null
4   P_min ← null
5   for all pools i = 1 : M do
6     R_i ← (w_i − T_i)/T_i
7     if R_i < R_min
8         R_min ← R_i
9         P_min ← i
10    else if R_i > R_max
11        R_max ← R_i
12        P_max ← i
13    end if
14  end for
15  if R_min < 0
16    switch from P_max to P_min
17  end if
```

Fig. 4. Pseudo code for the Average Response Time policy

a function, *percentInWindow* which given a response time target and window of most recent job response times evaluates the percentage of jobs which finished within the response time target.

4.3 QoS Policy 3: Holding Cost Weightings (HCW) policy

A method of provisioning servers based on QoS involves choosing the cost values of the dynamic allocation policy described in section 3 so that the resulting dynamic allocation of servers favours the pools with high QoS requirements at the expense of those with low ones.

The ratio of target response time to average service time can be taken as a "normalised" target. The smaller the normalised target is, the more difficult it is to meet, therefore the larger cost should be associated with it. Thus, the HCW policy is defined as the resource allocation heuristic of figure 2 with holding costs given by:

$$c_i = \frac{1}{\mu_i T_i} \tag{1}$$

where μ_i is the service rate for job type i and T_i is the target response time for job type i.

```
1   Per_max ← −∞
2   Per_min ← ∞
3   P_max ← null
4   P_min ← null
5   for all pools i = 1 : M do
6     Per_i ← percentInWindow(TarRT_i, Win_i)
7     if Per_i < TarPer_i
8         Per_min ← (Per_i − TarPer_i)
9         P_min ← i
10    else if Per_i > TarPer_i
11        Per_max ← (Per_i − TarPer_i)
12        P_max ← i
13    end if
14  end for
15  if Per_min < 0
16    switch from P_max to P_min
17  end if
```

Fig. 5. Pseudo code for the Percentile Policy

5 Implementation

A prototype system called GridSHED (Grid Scheduling Hosting Environment Design) has been developed. Each pool runs an instance of a Resource Management System (RMS) which queues, schedules and manages the execution of jobs. Most RMSs have a master-slave architecture where one host acts as the master of the cluster and is in control of many slave hosts.

All components in the system have been implemented using Java which was chosen because its cross-platform nature which meets the requirements of a heterogeneous data centre. A plugin architecture has been employed across the system which allows different implementations of core system components such as the Web service container, RMS, database and component communication system to be used which permits new and updated technology to be integrated into the system in a loosely coupled manner. In the implementation used for testing Java RMI was chosen for distributed component communication as it was deemed unnecessary to use Web services as all of the components are inside and under control of the same infrastructure, and Web services are designed for communication over organisational boundaries

The current implementation uses Condor [7] as the RMS but other RMSs may be wrapped in the GridSHED RMS interface and used as a queuing and scheduling system. Users may submit jobs made up of a binary program, input data and a submission script to a Condor enabled cluster where the jobs will be executed on a suitable server according to the requirements specified in the submission script.

A batch of jobs called a cluster may also be submitted which consist of one binary program and many different sets of input data which leads to many separate jobs with different parameters being submitted. A Condor pool has one central manager and a number of hosts and jobs may be submitted from any permitted host in a pool. A matchmaking mechanism called Class-Ads is used to match queued jobs to available resources, and results of an execution are sent back to the machine from which the jobs were submitted from.

Condor was chosen due to its flexibility and speed when switching servers between pools under its control. The system has also proved to be highly scalable despite the fact that all matchmaking takes place on the central manager. A disadvantage of Condor is that it does not have a notification system that could be used to listen for specific events, so all data obtained form Condor is polled at a set interval. Condor is cross platform allowing the Java based GridSHED system to run with no or little modification on all supported Condor platforms. The number of processors on a server can easily be obtained from Condor with exactly the same commands, regardless of the platform, which allows the number of processors in a pool to be used in the policies without the need for platform independent software

Condor is integrated with GridSHED software through a java API which allows job submission, queue querying, statistics gathering and general command execution. With the Condor interface the Pool Manager can submit jobs and query the queue, and Node Managers can reconfigure Condor daemons and gather statistics. New Condor hosts may be added and removed as necessary while the Condor central manager can probe the queue and continuously match jobs to available hosts un-interrupted by the addition and removal of servers. The Node Manager may configure the condor daemons running on its host to run in central manager mode which results in a new Condor pool for a newly deployed service.

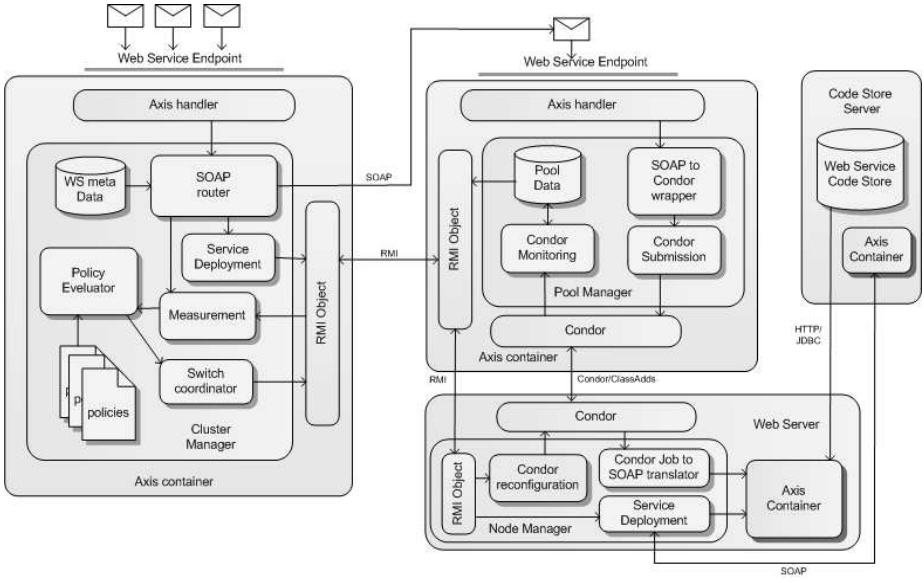


Fig. 6. System architecture components

To get state updates the Pool Manager polls Condor at a set interval and updates its current pool state if it has changed. The Cluster Manager then polls each Pool Manager at an interval and if there is any new state data it is returned to the Cluster Manager. This approach to polling was chosen as sometimes condor commands can block, and if a poll from the Cluster Manager led directly to a poll to Condor, the Cluster Manager could block which would slow down the system.

5.1 Web services integration

The Web services interface to the system via the Cluster Manager is implemented using hosted using Apache Axis [1] within the Apache Tomcat Web application container. All SOAP messages are sent to the Cluster Manager Web service although the endpoint URL used by the message sender appears to be the endpoint of the target service. Each SOAP message is intercepted by a global handler and directed to the Cluster Manager Web service, while the original service target specified by the user is saved in the SOAP header. The Cluster Manager then gets the address of the Pool Manager currently processing messages for the target service from its store of deployed services, then forwards the message to the Pool Manager.

When a SOAP message arrives at a Pool Manager the message is wrapped in a Condor job and submitted to Condor for queuing and scheduling. When the job is executed by Condor a simple client program sends the SOAP message to the service running on the local host (the deployment process ensures the service is installed on each server in a pool). Upon completion of SOAP message Execution the requests are sent back to the original invoker via the Pool Manager and Cluster Manager which allows data about service and total response times to be collected for each job.

A Web service Code Store is used to store Web service code that can be automatically installed on servers. Web services are packaged with any dependencies and uploaded to the code store. When a SOAP messages arrives for a service that is not deployed the Cluster Manager consults the Code Store to see if the service is available for deployment, an error is returned to the user if the Code Store knows nothing about the target service. If the service code is present in the Code Store, the Cluster Manager instructs the least loaded Pool Manager to reconfigure one of its servers to become a new Pool Manager capable of processing jobs for a newly deployed Web Service.

5.2 Server switching

A resource allocation policy will return a source and destination pool if a switch needs to be made. The Cluster Manager first sends a message to the source Pool Manager indicating it needs to reconfigure a server and the Pool Manager returns details about the server chosen to switch. The Pool Manager keeps the set of servers in its pool stored in a stack data structure so that servers that have been in the pool the longest are not switched out, which gives long-running jobs a chance to finish without having to suspend or check-point. The Cluster Manager may have

to re-calculate the switch decision if the source Pool Manager returns a server with more than one processor to make sure the switch is still viable, if it is the Cluster Manager instructs the source Pool Manager to go ahead with the switch. The Pool Manager instructs the Node Manager on the server to be switched to reconfigure by sending it the destination Pool Managers address and the name of the new Web service which is to be installed. When this is done the Cluster Manager is informed then the destination Pool Manager is sent the details of the reconfigured server which is then added to the pool.

6 Test data

In order to generate job submission patterns that are as close to the patterns received by a real job submission system, the usage of batch scheduling systems at our campus was studied. It was found that users tend to submit jobs in a session which consists of multiple consecutive clustered job submissions of up to thousands of jobs per cluster over a period of weeks, then users would not submit jobs for a number of months, presumably during which the results of job executions were analysed and acted upon. Individual job sizes range from 5 seconds to 2 minutes. These findings indicate that sessions are distributed in a uniform distribution, as are the execution times of jobs.

In the system described, jobs, or SOAP messages, cannot be submitted as a cluster as in a standard job submission system such as Condor. However the same effect can be achieved in a less efficient manner by sending a high volume of SOAP messages in a short time, although SOAP messages will only contain parameters and data and not binary files as in a job cluster.

7 Experimental Results

The QoS policies listed in section 4 were each separately employed in different experiments under the same workload to measure the percentage of jobs that completed within their target response time. Three Web services were deployed on a cluster of 8 servers each containing 4 processors, giving a total of 32 processors. The Web services made arbitrary mathematical calculations to generate load upon invocation, the time each service performed calculations for was specified in the SOAP message sent to the service. Servers were switched between each of the service pools when instructed by the QoS policy in place, and the implementation required that at least one server always had to be in a pool. Two sets of experiments were performed for each of the three QoS policies, one used strict response time targets and the other used more relaxed targets to observe how performance differed under stricter targets. In each experiment a total of 1000 jobs was sent via three different job streams, one for each deployed Web service. Jobs, or SOAP messages, were submitted at a given average arrival rate which was programmatically altered according to a job submission pattern in an attempt to simulate real-world jobs submission patterns. Two different arrival rate determination policies are used.

`The Session Job Submission policy` set the arrival rates according to the job submission pattern described in section 7. To test the system within a reasonable time frame the observed job submission pattern was scaled down so the time between sessions was minutes rather than months, and submission sessions consisted of a high job arrival rate for a number of seconds. The average values for the arrival interval and job time were all determined using the uniform distribution, which appeared to match the observed behaviour from real users whose job times, time between sessions and arrival rate intervals during a submission session had upper and lower bounds. The stream then sent no jobs for the calculated wait time then sent jobs at the specified arrival rate for the given submission time.

`The Phased Job Submission policy` sent a constant stream of jobs at varying phases of arrival rates in order to simulate a service which receives a continual load. The phases were changed according to a phase window which consisted of x jobs, where x was set in a configuration file. The arrival intervals were determined by the exponential distribution which captured the arrival pattern of multiple independent parties.

Two job types were submitted using the The Session Job Submission policy and one submitted using the Phased Job Submission policy. One can deduce that the response time target will be harder to meet for Web services receiving jobs sent using the Session policy because the last batch of jobs submitted will wait on a longer queue for longer periods of time than jobs submitted using the Phased policy. This suggests that the two job types sent via the Session policy should have higher target response times than job sent via the Phased policy.

In the first set of test the QoS targets were relaxed and in the second they were strict. The target response time for the Web services receiving jobs via the Session Submission Policy was set to 10 times the specified average job time. The target response time for the Web service receiving jobs using the Phase Job Submission policy was set to 5 times the average job time as the jobs would naturally finish quicker. In the Second set of experiments the target response time for Web services receiving jobs using the Session Job Submission policy was set to 5 times the average job length and the target response time for the Web service receiving jobs using the Phase Job Submission policy was set to 2.5 times the average job length. Each job type had the same arrival rate pattern and job service in each test; these details are shown in the tables below. Figure 7 shows the phase-based job submission characteristics of job type 1, and Figure 8 shows the session-based job submission characteristics of job types 2 and 3. Figure 9 Shows the average offered load of each job type in the experiments. The total offered load of the 32 processors is 80.12%.

Phase	Arrival rate (jobs per second)	Job Time (Seconds)	phase window (jobs)
1	0.034	180	100
2	0.0167	180	100
3	0.0167	180	100

Fig. 7. Phased Job Submission policy parameters of job type 1

Job type	Arrival rate (jobs per second)	Job length (Seconds)	Submit time (Seconds)	Session interval time (Seconds)
2	1	150	60	500
3	1	30	60	500

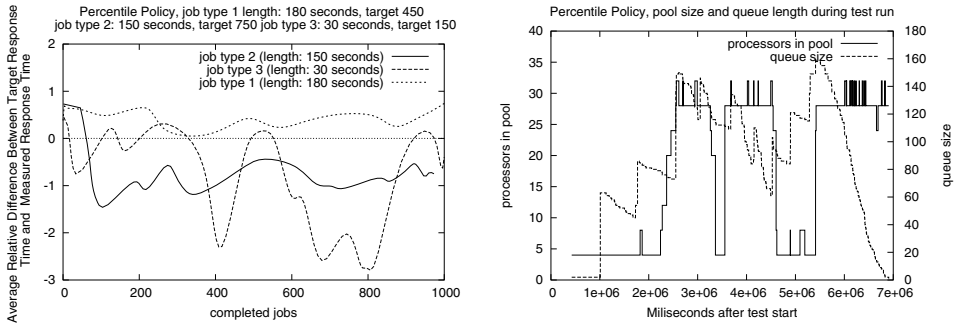
Fig. 8. Session Job Submission policy parameters of job types 2 and 3

Job Type	Average Arrival rate (jobs per second)	Job Time (Seconds)	Offered load
1	0.022	180	4.04
2	0.12	150	18
3	0.12	30	3.6
		total:	25.64

Fig. 9. Average offered load

7.1 Strict response time target test results

The following results are for the stricter target response time tests. Figure 10 shows the results for the Percentile policy, Figure 11 shows results form the Average response time policy, and Figure 12 shows the results from the QoS heuristic policy.



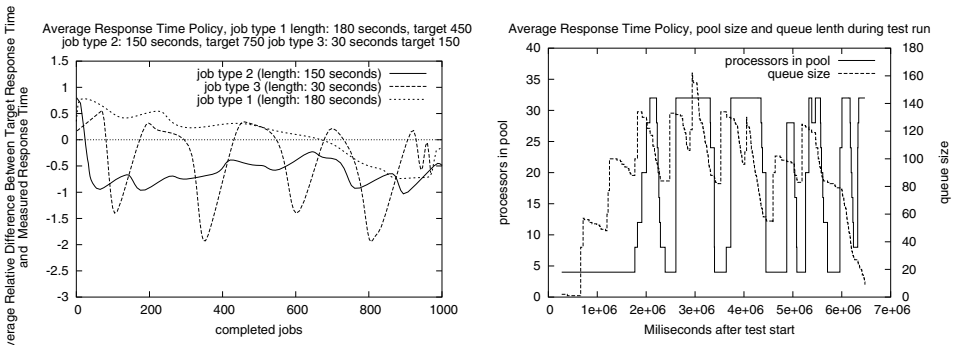
(a) (b)

Fig. 10. Percentile policy test measurements with a strict targets

The Percentile policy missed most of the targets, and the pool serving job type 1 was allocated servers only after the queue rose to a significant level. This policy also appears to have lead to a lot of unnecessary switching, as shown by the constant fluctuations of pool size in figure 10(b).

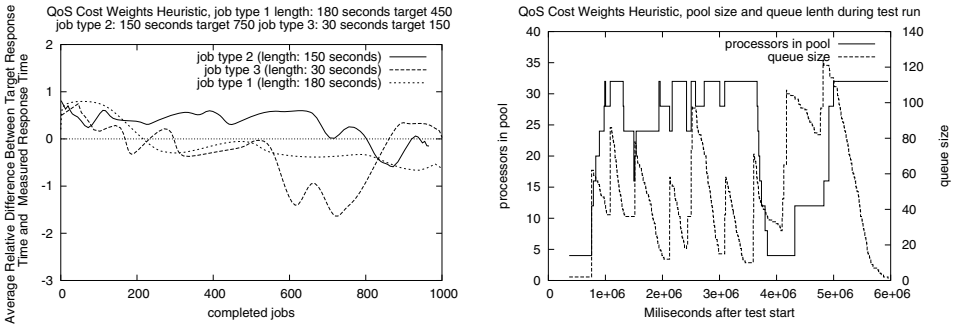
The ratio based switching method also missed most of the targets, but the servers did not switch as much as the percentage based switching policy which is good for system efficiency.

The HCW policy appears to have performed better than the other policies and servers appear to have been allocated to the pool serving job type 1 just as the queue was increasing. figure 13 shows the percentage of jobs which matched their target arrival rate; the HCW policy has the highest number of matched jobs, with the Percentile policy performing worse while the Average response time policy performed worse still. The HCW policy performed worse than the other two policies for jobs of type 1 which was the job type submitted using the Phased Submission



(a) (b)

Fig. 11. Average Response Time policy test measurements with strict targets



(a) (b)

Fig. 12. Holding Cost Weightings policy test measurements with strict targets

Pattern policy. It can be seen in figure 12(a) that job type 1 rarely had a relative difference between measured and target response time of below -0.5, so most jobs were not as late the jobs failed by the other polices.

Job Type	Percentile Policy	Average Response Time Policy	Heuristic Policy
1	86.3%	59.2%	31.3%
2	6.5%	7.1%	75.6%
3	33.8%	44.8%	41.5%
Average percent:	42.2%	37.0%	49.5%

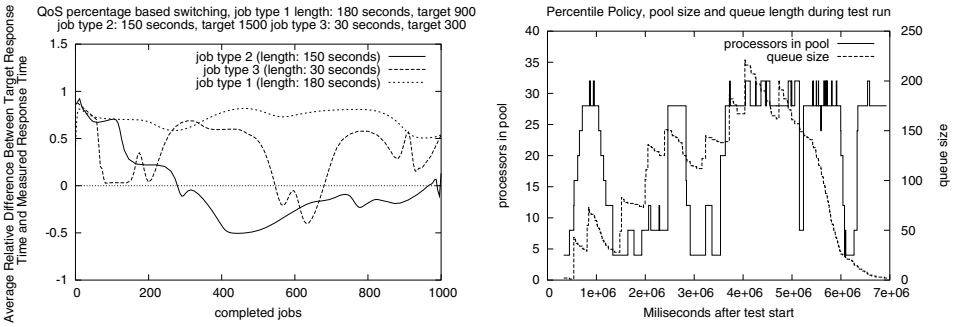
Fig. 13. The percentage of jobs which matched their target arrival rate with strict targets

7.2 Relaxed response time target test results

The next sets of results are for the experiments performed with more relaxed target response times so one would expect all of the policies to perform better than they did with stricter targets.

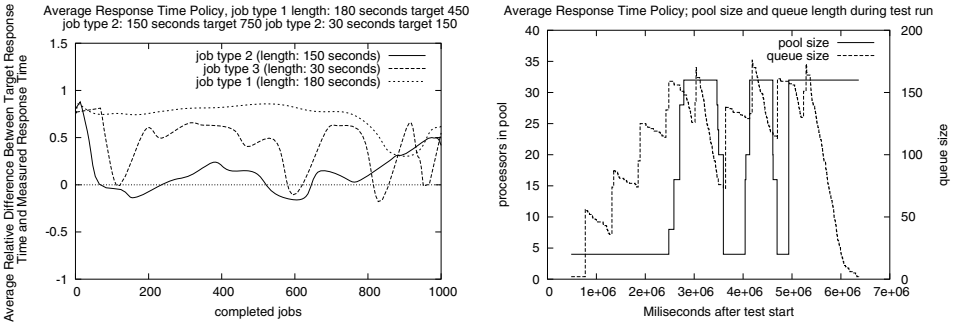
The Percentile policy has performed better with a relaxed target although one of the job types has failed to meet its target for over 75% of the test time.

The Average Response Time policy has performed better than the Percentile



(a) (b)

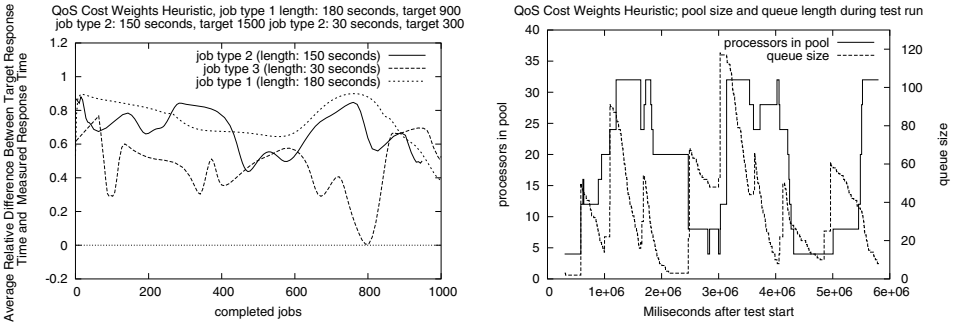
Fig. 14. Percentile policy test measurements with a relaxed target



(a) (b)

Fig. 15. Average Response Time policy measurements with relaxed target

policy with a relaxed target, and while the servers seemed to have switched pools in a smoother fashion the queue size had to rise before servers started switching into the pool.



(a) (b)

Fig. 16. Holding Cost Weightings policy test measurements with strict target

The HCW policy ensured all jobs met their target response times, and servers switched in time to process incoming jobs.

Figure 17 shows the percentage of jobs which matched their target arrival rate

Job Type	Percentile Policy	Average Response Time Policy	Heuristic Policy
1	98.9%	98.9%	100%
2	40.1%	67.3%	100%
3	75.8%	84.8%	100%
average percent	71.6%	83.7%	100%

Fig. 17. The Percentage of jobs which matched their target arrival rate with relaxed targets

with relaxed target response times. This time the Percentile policy is worst which appears to indicate the Percentile policy performs better under stricter response time targets.

8 Conclusion

We have shown how a stochastic model for resource allocation may be applied to a Web service hosting environment which incorporates QoS for job response time. Different approaches to Quality of Service were compared and the method of setting the costs of the stochastic model based on response time targets proved to be the best approach.

The experiments performed showed that the Cost Weightings policy was able to allocate servers more effectively to cope with demand and satisfy response time targets than the Average response time and Percentile policy. The Average response time and Percentile policies did not perform well because their server switching decisions were based on the response times of jobs which had completed in the past, so when a large queue built up at a pool, the policies did not switch servers fast enough to cope with the demand and only switched servers after a number of jobs had failed. Worse still, after a session of jobs had completed at a pool and all of the jobs had been processed, the pool had a high average response time. This meant that as a session started at a another pool servers did not switch as the previous pool had a far higher average response time and the Percentile and the Average Response Time policy therefore did not switch any servers. This showed the Percentile and Average Response Time policy were especially bad when dealing with abrupt changes in demand, such as those received by batch scheduling systems.

References

[1] Apache Axis, http://ws.apache.org/axis/.

[2] Atkinson M, DeRoure D, Dunlop A, Fox G, Henderson P, Hey T, Paton N, Newhouse S, Parastatidis S, Trefethen A, Watson P, Webber J, "Web Service Grids: An Evolutionary Approach," *UK e-Science Technical Report Series*, http://www.nesc.ac.uk/technical_papers/UKeS-2004-05.pdf, 2005.

[3] Chandra A, Gong W, Shenoy P, "Dynamic Resource Allocation for Shared Data Centers Using Online Measurements," *Presented at Eleventh International Workshop on Quality of Service*, 2003.

[4] Chase JS, Irwin DE, Grit LE, Moore JD, Sprenkle SE, "Dynamic Virtual Clusters in a Grid Site Manager," *presented at the 12th IEEE International Symposium on High Performance Distributed Computing*, 2003.

[5] Fisher M, Kubicek C, McKee P, Mitrani I, Palmer J, Smith R, "Dynamic Allocation of Servers in a Grid Hosting Environment," *Presented at the Fifth IEEE/ACM International Workshop on Grid Computing*, 2004.

[6] Kubicek C, Fisher M, McKee, P, and Smith, R, "Dynamic Allocation of Servers to Jobs in a Grid Hosting Environment," *BT Technology Journal, vol. 22, pp. 251-260*, 2004.

[7] Litzkow M, Livny M, Mutka M, "Condor - A Hunter of Idle Workstations," *Presented at 8th International Conference of Distributed Computing Systems*, 1988.

[8] Menasce DA, Barbara D, Dodge R, "Preserving QoS of E-commerce Sites Through Self-Tuning: A Performance Model Approach," *Presented at the ACM Conference on E-commerce*, 2001.

[9] Mitrani I, Palmer J, "Dynamic Server Allocation in Heterogeneous Clusters," *Presented at The First International Working Conference on Heterogeneous Networks, Bradford*, 2003.

[10] Ranjan S, Rolia J, Fu H, Knightly E, "QoS-Driven Server Migration for Internet Data Centers," *resented at Tenth International Workshop on Quality of Service* 2002

[11] W3C, "SOAP Version 1.2 Part 1: Messaging Framework," *ed. M Gudgin, M Hadley, N Mendelsohn, J-J Moreau, HF Nielsen* 2003.

[12] Watson P, Fowler C "An Architecture for the Dynamic Deployment of Web Services on a Grid or the Internet," *Technical report, University of Newcastle upon Tyne, UK* 2005.